Transformations and Software Modeling Languages: Automating Transformations in UML

Jon Whittle

QSS Group Inc.
NASA Ames Research Center
Moffett Field
CA 94035
jonathw@email.arc.nasa.gov

Abstract. This paper investigates the role of transformations in the Unified Modeling Language, specifically UML class diagrams with OCL constraints. To date, the use of transformations in software modeling and design has not been fully explored. A framework for expressing transformations is presented along with concrete examples that, for example, infer new inheritance links, or transform constraints. In particular, a technique for checking that two UML class diagrams are refactorings of each other is described.

1 Introduction

Transformations play a central role in software engineering. They have found widespread use in traditional applications such as optimizing compilers, as well as in contemporary activities such as the use of XML and XSLT to translate between different data formats. To date, however, the use of transformations in practical software *modeling* has been limited (cf. [3, 9, 1] for the state of the art). This paper begins to redress this imbalance by presenting a framework for representing transformations of modeling notations along with examples of transformations and their applications. There are a number of potential starring roles for transformations. In particular, the widespread use of the Unified Modeling Language (UML) [10] provides an immediate user base for model transformation techniques. This paper focuses on applications of transformations within UML (as opposed to transformations from UML to another language). Possible roles for transformations include the following:

1. model optimizations (transforming a given model to an equivalent one that is optimized, in the sense that a given metric or design rule is respected);
2. consistency checks (transforming different viewpoints of the same model into a common notation for purposes of comparison — cf. [12] for an example of consistency between UML sequence diagrams and UML's constraint language OCL);
3. automation of parts of the design process (cf. [2] for recent work in this field) using transformations.

The paper will give an extended example of the use of transformations in an application of type (3) above. The example concerns the use of transformations to show that two class diagrams (with additional constraints) are refactorings of each other. Refactoring is the process of redesigning the structure of a software artifact without changing its behavior. It is usually applied to code to make it more readable or reusable, but can just as usefully be applied to models of the code. Indeed, it could be argued that refactoring is more important at the modeling level since restructurings are very common at this stage of the development process. This paper presents a technique for checking refactorings, along with an empirical example that demonstrates its use in practice.

Initial implementation of the transformation framework was done in the logic-based programming language Maude [5]. Maude code consists of a set of equations and rewrite rules. The Maude execution engine applies these rules to transform a given term and can be tailored to particular execution strategies using Maude's meta-level. Maude provides a suitable environment for experimenting with and automating the application of transformation rules in UML. An implementation of the abstract syntax of UML already exists as a set of Maude theories [11]. This implementation faithfully mirrors the UML meta-model and is a suitable basis for implementing UML-based transformations. Maude proved to be a suitable environment for experimenting with transformation rules.

There have only been a small number of papers that directly deal with model-level transformations in UML, and most of these have been concerned with the formal semantics of the transformations and proofs that they maintain correctness. [6] presents a set-theoretic formalization of what it means for one class diagram to be a logical consequence of another: a well-formed diagram D' follows from a well-formed diagram D if and only if every set assignment that satisfies D also satisfies D'. A well-formed class diagram is one that conforms to the UML specification [10]. A set assignment is the assignment to each class a set of object instances, and to each association a set of links between object instances. Informally, a set assignment then satisfies a class diagram, D, if the assignments of objects and links respects the semantics (according to the UML specification) of each of the class and association specifications in D. Although the issue of correctness of transformations is an important one, it is not considered in this paper. Instead, the emphasis is on automation of transformation applications. Automation means both implementing the transformation rules so that they can be executed and automating the application of sequences of transformation rules, which can be carried out by grouping rules together into strategies and searching through the possible applications of the strategies to achieve some goal. Although other work (e.g., [1, 9, 6]) has given examples of transformations, no attempt has yet been made to automate the construction of sequences of transformations that carry out a particular task.

2 Transformations in UML

2.1 Definitions

This paper will limit discussion to UML class diagrams annotated with OCL constraints. In addition, advanced features of class diagrams (such as interfaces, parameterized classes, etc.) will be ignored. In principle, many of these advanced features can be described in terms of the more basic notions given (see [8]). A UML class diagram is represented by a ternary mixfix operator:

$$Classes \ \# \ Associations \ with \ Constraints$$

where $Classes$ may represent either a UML class or type[1] and constraints are general restrictions on the class diagram that will be given as OCL constraints in this paper. $Classes$, $Associations$ and $Constraints$ are each sets whose elements are formed from the following constructors:

$$\begin{array}{l}\mathbf{class}(className, Attributes, Operations)\\ \mathbf{association}(assocName, assocEnd_1, assocEnd_2)\\ \mathbf{oclConstraint}(expression, modelElement, stereotype)\end{array}$$

where $className$ and $assocName$ are drawn from the set of class names and association names, $ClassNames$ and $AssocNames$, respectively. There are similar sets, $AttrNames$ and $OpNames$, to hold the names of attributes and operations. Sets will be denoted using identifiers with upper case initial letters. Where necessary, elements of sets will be written down separated by whitespace. Each class has a set of attributes and a set of operations. Each constraint is formed from an OCL expression, a model element that is the context of the constraint, and a stereotype that denotes the kind of constraint (e.g., invariant, pre-condition). Each association has two association ends, $assocEnd_1$ and $assocEnd_2$, represented as follows[2]:

$$\mathbf{assocEnd}(className, Multiplicity, assocType, navigable)$$

The association end is attached to the classifier $className$. $Multiplicity$ is a set of (possibly unbounded) integers denoting the possible number of instances of $className$ associated with each instance of the classifier at the opposite end of the association. $assocType$ is either gen, agg, $comp$ or $assoc$ depending on whether the association end is a generalization, aggregation, composition or regular association, respectively. $navigable$ is a boolean value denoting whether or not the association is navigable towards this association end. Note that generalization relations are only navigable in one direction (from child to parent) and so such associations will always have false navigability from parent to child.

[1] throughout, class will be used to mean either class or type

[2] Association ends may also have orderings, qualifiers, rolenames, interface specifiers, changeabilities and visibilities (see [10], page 3-71) but these are omitted in this paper.

This formalization has been implemented in the logic-based programming language Maude by researchers at the University of Murcia [11]. The transformation rules presented in the next section were also implemented in Maude using this formalization as a basis.

2.2 Transformation Rules

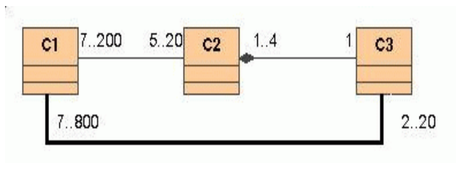
This section presents a number of examples of transformation rules on UML class diagrams with constraints.

Transitivity of associations The same model structure can be represented in many different ways using a class diagram. Transformations that derive new model elements (called enhancement transformations in [9]) augment a given diagram with additional classes, associations or constraints that can be logically inferred from the current structure. Enhancement transformations can be used to make explicit design documentation that may have been left hidden by the designer.

$$
\begin{array}{l}
association(assocName_1, assocEnd(c_1, M_1, a_1, v_1), assocEnd(c_2, M_1', a_1', v_1')) \\
association(assocName_2, assocEnd(c_2, M_2, a_2, v_2), assocEnd(c_3, M_2', a_2', v_2')) \\
\Longrightarrow \\
association(newAssocName, \\
\quad assocEnd(c_1, mult_1(M_1, M_1', M_2, M_2'), ass_1(a_1, a_1', a_2, a_2'), nav_1(v_1, v_1', v_2, v_2')), \\
\quad assocEnd(c_3, mult_2(M_1, M_1', M_2, M_2'), ass_2(a_1, a_1', a_2, a_2'), nav_2(v_1, v_1', v_2, v_2'))) \\
association(assocName_1, assocEnd(c_1, M_1, a_1, v_1), assocEnd(c_2, M_1', a_1', v_1')) \\
association(assocName_2, assocEnd(c_2, M_2, a_2, v_2), assocEnd(c_3, M_2', a_2', v_2'))
\end{array}
$$

Fig. 1. Transitivity of Associations: *transitivityAssocs*.

Figure 1 shows an example of an enhancement transformation that infers new associations in a class diagram using the transitivity property of associations. Transformation rules should be interpreted such that if any part of a class diagram is able to match with the LHS of the rule, then the rule is available to fire. On firing, the sub-diagram matching with the LHS is replaced by the RHS with the appropriate instantiations of variables. A rule that derives new associations must infer the classes to which the association is attached as well as the multiplicities and association types at each end of the association. The rule in Figure 1 says that given an association between c_1 and c_2 and an association between c_2 and c_3, an association can be derived between c_1 and c_3. $mult_1$ and $mult_2$ calculate the appropriate multiplicities at each end of the new association. ass_1 and ass_2 determine the association types at each new association end.



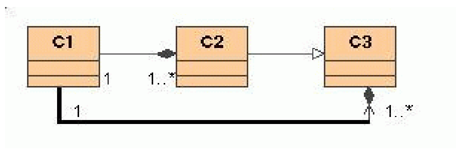


Fig. 2. (a) An application of transitivity of associations with bi-directional navigability (b) An application of transitivity of associations with uni-directional navigability

$$ass_1(a_1, a_2, a_3, a_4) = a_1 \times a_3$$
$$ass_2(a_1, a_2, a_3, a_4) = a_2 \times a_4$$

$\times$	gen	agg	comp	assoc
gen	gen	agg	comp	assoc
agg	agg	agg	agg	assoc
comp	comp	comp	comp	assoc
assoc	assoc	assoc	assoc	assoc

Fig. 3. Composing association types.

nav_1 and nav_2 calculate the navigabilities. Figure 2(a) shows an example of a rule application where the association in bold is a derived association. Since each instance of C_2 has an association to instances of C_1 and each instance of C_3 is part of instances of C_2, then each instance of C_3 must be related to instances of C_1. Hence, an association can be derived between C_1 and C_3. The appropriate multiplicities on this derived association are calculated using an algorithm in [1]. The algorithm presented there computes new multiplicities for a derived association between classes c and c' based on a traversal of an existing path from c to c' that collects and merges multiplicity information along the way. The algorithm from [1] is called as a subroutine by the transformation rule in Figure 1.

Figure 3 shows how to derive association types for a newly inferred association. In Figure 2(a), $ass_1(a_1, a_1', a_2, a_2') = ass_1(assoc, assoc, comp, assoc) = assoc \times comp = assoc$ and $ass_2(a_1, a_1', a_2, a_2') = assoc \times assoc = assoc$. Note that for some combinations of adjacent associations, only uni-directional associations can be inferred. As an example, in Figure 2(b), given the associations between C_1, C_2 and C_2, C_3, an association between C_1 and C_3 can be inferred as shown, but this association is not navigable from C_3 to C_1. Navigability from C_1 to C_3 is legal because each instance of C_1 is part of an instance of C_2, and hence is part of an instance of C_3. In the opposite direction, however, an instance of C_3 need not also be an instance of C_2 and hence may not have an association to C_1. Navigability is calculated in a similar fashion to association types except that the $\times$ operator is replaced by boolean $\wedge$:

$$nav_1(v_1, v_2, v_3, v_4) = v_1 \wedge v_3$$
$$nav_2(v_1, v_2, v_3, v_4) = v_2 \wedge v_4$$

In Figure 2(b), $nav_1(v_1, v_1', v_2, v_2') = true \wedge false = false$ and $nav_2(v_1, v_1', v_2, v_2') = true \wedge true = true$, hence the derived association (in bold) has navigability only towards C_3.

The transitivity rule is similar to an example transformation given in [7] except that it generalizes the notion of transitivity to all association types.

Introducing subclasses A much simpler transformation rule is given in Figure 4. This rule introduces a new subclass, *child*, of *parent* that satisfies additional constraints Ψ. Note that Ψ need bear no relation to Φ since they are new (possibly empty) constraints that characterize the new subclass. The new subclass *child* inherits the attributes, operations and constraints of its parent (an alternative would be for another rule to take care of propagating the parent constraints to the child).

$$\begin{array}{l}
class(parent, Attrs, Ops) \ \# \ Associations \\
\qquad\qquad with \ oclConstraint(\Phi, parent, \ll \texttt{invariant} \gg) \\
\Longrightarrow \\
class(parent, Attrs, Ops) \\
class(child, Attrs, Ops) \ \# \\
association(newAssocName, assocEnd(parent, \{1\}, gen, true), \\
\qquad\qquad\qquad\qquad assocEnd(child, \{0,1\}, assoc, false)) \\
Associations \ with \\
oclConstraint(\Phi, parent, \ll \texttt{invariant} \gg) \\
oclConstraint(\Phi, child, \ll \texttt{invariant} \gg) \\
oclConstraint(\Psi, child, \ll \texttt{invariant} \gg)
\end{array}$$

Fig. 4. Introduction of Constrained Subclasses: *introduceSubclass*.

$Classes \ \# \ assoc \ Associations$

$with \ oclConstraint(\textbf{if} \ cond \ \textbf{then} \ exp_1 \ \textbf{else} \ exp_2 \ \textbf{endif} \ , class, st) \ Constraints$

$\Longrightarrow$

$Classes \ \# \ assoc' \ Associations \ with \ Constraints$

if *cond* holds in *class*

and *exp* can be eliminated by replacing *assoc* with *assoc'*

Fig. 5. Inferring associations from constraints: *inv2assoc*.

Relating constraints and associations Transformations on class diagrams with constraints must relate class diagram elements and constraint expressions. Constraints may express information that could otherwise have been (partially) expressed using classes and associations. Transformations that translate between these alternative viewpoints would be useful in model optimization. As an example, OCL constraints on the number of instances of a particular class could also be expressed using association multiplicities. If, in fact, both are used, it is possible either that the two are in conflict with each other or that one is subsumed by the other. In the latter case, eliminating the subsumed constraint or association will lead to generation of more efficient code since the redundancy is removed.

Figure 5 gives an example transformation rule for relating constraints and UML associations by updating an association according to the constraints. The rule is conditional, in the sense that the rule only fires if the two conditions associated with it hold. The rule shows how a conditional OCL constraint can be eliminated by updating the appropriate association. The idea is that if the condition *cond* holds in the class which owns the constraint, then exp_1 must be true, and, in some cases, the truth of exp_1 can be used to modify the association *assoc* into *assoc'*. In the form of the rule given here, the rule only fires if the conditional constraint can be completely rewritten as an association *assoc'*. In general, however, only part of a constraint may be eliminated.

As a concrete example of a rule application, in Figure 6, each instance of class C_1 can be related to either zero or two instances of C_2. Suppose that there is an additional OCL constraint on class C_1:

```
context C1  inv:
if Cond  then itsElements->size=2
       else itsElements->size=0  endif
```

If it can be shown that `Cond` holds as an invariant in C_1 (e.g., if it is satisfied by some other constraint of C_1), then it can be deduced that each instance of C_1 must associate with exactly two instances of C_2 and so, the multiplicity of `itsElements` can be updated to read 2 rather than $0, 2$. Similarly, there is another transformation rule that applies in the case that *cond* does not hold. In this case, the multiplicity could be updated to read 0. Note, however, that the latter case relies on the closed world assumption — i.e., failure to show *cond* is an assumption that *cond* is false.

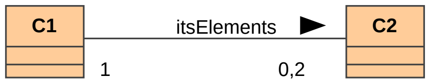


Fig. 6. Updating Multiplicities according to Constraints.

Clearly, relating constraints to model elements requires some non-trivial inferences in general. In particular, in this example, it needs to be shown that *cond* holds in *class* and more generally, it may be useful to make non-trivial inferences about how exp_1 relates to the association being updated. The level of sophistication required depends on the application. For a model optimization tool, the approach followed in optimizing compilers of making many simple inferences but not attempting more complex ones will work well. For some applications, however, more complex inferences may be required. Further work will investigate the use of "off-the-shelf" decision procedures for fragments of first-order logic, such as the Stanford Validity Checker [4], which could be useful for dealing with a range of complex subproblems expressed in OCL.

Inferring new generalizations Previously, it was shown how new associations (including generalizations) can be inferred by composing existing associations. Another way of deriving new generalizations is to compare the constraints, associations, attributes and operations of two existing classes (see Figure 7). Given classes *parent* and *child*, if it can be shown that all the constraints of *parent* hold in *child* and similarly, for the associations, operations and attributes of *parent*, then a new generalization association can be inferred from *child* to *parent*. In Figure 7, *constraints* is an operation that, for a given set of constraints (indicated by its argument), returns the subset of those constraints that apply to the class to which the operation is attached. Similarly for the operation *associations*. Comparison of constraints and associations is syntactical comparison of sets. Hence, the rule is rather weak in the sense that a constraint of *parent* may actually hold in *child* but may not be in the same syntactical form. In this case, for the rule to apply, the constraint must first be transformed into the same form. In general, this may involve non-trivial reasoning.

Three other transformation rules will be mentioned because they will be used as part of the example in the following section. *inheritConstr*, *inheritAttr* and *inheritOp* are transformations that propagate constraints, attributes and operations, respectively, from a parent class to its children classes. They are simple transformations that impose the class diagram restrictions that a child

$$class(parent, Attrs_1, Ops_1)\ class(child, Attrs_2, Ops_2)\ \#\ Associations\ with\ Constraints$$

$$\Longrightarrow$$

$$class(parent, Attrs_1, Ops_1)\ class(child, Attrs_2, Ops_2)\ \#$$
$$association(newAssocName, assocEnd(parent, \{1\}, gen), assocEnd(child, \{0, 1\}, assoc))$$
$$Associations\ with\ Constraints$$

if $parent.constraints(Constraints) \subseteq child.constraints(Constraints)$
and $parent.associations(Associations) \subseteq child.associations(Associations)$
and $Attrs_1 \subseteq Attrs_2$
and $Ops_1 \subseteq Ops_2$

Fig. 7. Inferring generalizations: *inferGen*.

class must have (at least) the same constraints, attributes and operations as its parent.

3 Checking Model Refactorings

Refactoring is the process of restructuring an existing software artifact whilst maintaining its behavior. Refactoring has usually been considered only at the code level, but, in fact, refactoring is just as important during the design phase. Early design work typically involves iterative updates of the design at hand and restructurings are often a necessary prerequisite for these updates to take place. The main problem with refactorings is that it is easy to introduce design bugs. This section shows how transformations can be used to check that a modified design is a refactoring of an existing one, and hence show that no unforeseen errors have been introduced. A technique will be presented for automatically checking that two UML class diagrams are refactorings of each other. The technique is illustrated using the example class diagrams of Figures 9 and 10 and the transformations from the preceding section.

3.1 Checking Procedure

A procedure for checking if two class diagrams D_1 and D_2 are refactorings of each other is now presented. This procedure is based on *difference matching*, i.e., looking for differences between D_1 and D_2 and applying transformation rules so as to eliminate them. Some transformation rules (e.g., *inheritConstr* and *inheritAttr*) can be applied exhaustively to a class diagram as they do not involve choice points that would lead to a large search space. Rules such as *introduceSubclass*, however, are akin to the cut rule in first order logic proof systems in that, when applied to a single class diagram, they require a "eureka step" to instantiate an unbound variable. Difference matching provides the means to avoid these eureka steps.

There are three main definitions that form the checking procedure: the definition of a mapping, ϕ, to map the elements of D_1 into those of D_2, the definition of difference matching to translate $\phi(D_1)$ into D_2 and the definition of strategies for automating the rule applications involved in difference matching. Defining strategies involves the development of heuristics to reduce the search space. This is not dealt with in this paper.

Mapping Model Elements During a restructuring operation, the names of model elements may be changed. Since this should not effect the validity of the refactoring, however, the model element names (i.e., the association, class, attribute and operation names) of D_1 are first mapped into those of D_2. For the example presented in this paper, it is assumed that there have been no renamings and so the mapping, ϕ, is just the identity mapping. More generally, however, ϕ could be explicitly given by the user by specifying relationships between model elements using UML's $\ll$ *trace* $\gg$ stereotype. The $\ll$ *trace* $\gg$ stereotype denotes a historical connection between two model elements that represent the same concept at different levels of meaning. Hence, a $\ll$ *trace* $\gg$ stereotype could be used to connect concepts that have been renamed. The construction of ϕ could also be (partially) automated by a tool that keeps track of renaming of model elements and inserts the appropriate $\ll$ *trace* $\gg$ relationships.

Difference matching A distinction shall be made between transformation rules that can be applied exhaustively without fear of search space explosion (*normal* rules) and those rules whose application must be controlled through the use of heuristics (*non-normal* rules). Of the rules presented in this paper, *inheritConstr*, *inheritAttr* and *inheritOp* fall into the normal category while all other rules are non-normal. Non-normal rules are applied only in the context of a particular application using heuristics that have been designed for that application. As an example, consider *introduceSubclass* from Section 2.2. In order for this rule to apply, the variable *child* on the RHS of the rule must be instantiated with the name of a new subclass and Ψ must be instantiated with a set of constraints which hold over this new subclass. Clearly, exhaustive application of this rule introduces an infinite search space. Difference matching can be used, however, to choose the instantiations for *child* and Ψ, by comparing the source and target class diagrams and introducing new *child* subclasses to make the source and target match.

Another rule for which difference matching is required is *transitivityAssocs*. Exhaustive application of this rule does not result in an infinite search space but does introduce up to $n^2 + n$ new associations for a class diagram with n classes. This is because, by definition of a class diagram, each class has at least one association to some other class, c, and hence, by transitivity, there is an association to all classes that can be reached from c. For large class diagrams, the number of derived associations can become unmanageable and so difference matching is used to only introduce new associations as necessary.

```
Input. Class diagrams $D_1$ and $D_2$
• Apply all normal rules to $\phi(D_1)$ and $D_2$
• Match the class structure of $\phi(D_1)$ and $D_2$:
  for each class $c \in \phi(D_1)\backslash D_2$
          find a set of rules $\mathcal{R}$ that introduces $c$ into $D_2$
          (similarly, for $c \in D_2\backslash\phi(D_1)$)
          Apply the rules in $\mathcal{R}$ to $D_2$
          Apply all normal rules exhaustively to $D_2$
• Match the associations in $\phi(D_1)$ and $D_2$:
          find and apply $\mathcal{R}$ as above and
          apply all normal rules exhaustively
• Match the constraints in $\phi(D_1)$ and $D_2$:
          find and apply $\mathcal{R}$ as above and
          apply all normal rules exhaustively
• repeat
```

Fig. 8. Difference matching class diagrams.

An outline of difference matching is given in Figure 8. Matching classes amounts to transforming the source class into a new class with the same attributes, operations and constraints as the target class. Matching associations amounts to transforming the source association into a new one with the same association end class names, multiplicities, navigabilities and association types as the target association. Note that $\mathcal{R}$ may contain both normal and non-normal rules, but after application of the rules in $\mathcal{R}$, all normal rules are applied (which can be done efficiently) to propagate as much information as possible given the new model elements that have been introduced. For example, if a new subclass has been introduced, normal rules will be applied to make the subclass inherit attributes and operations from its parent.

Clearly, the procedure in Figure 8 requires a judicious control of applications of the rules in $\mathcal{R}$. The optimal control in the case of the refactoring application is the subject of current research and in the example which is about to be presented, the key choice points were decided manually. However, the split of rules into normal and non-normal rules already provides a good amount of control. Normal rules can be applied without user interaction. It is sometimes necessary, however, to impose an ordering on multiple applications of the same normal rule. As an example, the *inheritAttr* rule must be applied in the order of the inheritance graph — i.e., the top-level parent class is first chosen to be the root of this graph and the child is its first subclass, so that the first application of the rule propagates attributes from the root parent to its first child. Subsequent instantiations of parent and child are chosen by following a depth-first search over the inheritance structure. This ordering ensures that all attributes are included in all appropriate subclasses.

3.2 Example

An example of checking refactorings will now be presented. The example comes from design documentation for a research prototype under development at NASA Ames Research Center for generating code for avionics applications. The design subset considered in this paper is part of a domain model for matrix factorizations. Matrix factorizations are important for avoiding round-off errors in geometric state estimation algorithms (i.e., algorithms that estimate the state – position, velocity, etc. – of a vehicle given noisy sensor measurements). Various standard factorization tricks can be used to ensure a numerically stable implementation of a state estimation algorithm. Two examples are decomposition of a matrix into cholesky factors or rank one factors. A cholesky factor of a symmetric nonnegative definite (SND) matrix M is a matrix C such that $CC^T = M$, where C is triangular. A rank one factor is a cholesky factor of the rank one modified matrix $M + vv^T$ where v is a vector. Cholesky factors only exist for SND matrices and rank one factors only exist for symmetric positive definite (SPD) matrices. These restrictions are captured by constraints Φ_1 and Φ_2 in Figure 11.

Figure 9 (along with constraints in Figure 11) shows the initial attempt at a domain model for matrix factorizations. As part of a refinement, it was decided that Figure 10 (constraints in Figure 12) would serve as a better model for the domain. Additional matrix classes have been added and given appropriate OCL constraints. The addition of classes for SND and SNP matrices allows Φ_1 and Φ_2 to be represented directly as multiplicity constraints on the relevant associations and so Φ_1 and Φ_2 do not apply to Figure 10 (note, however, that Φ_3 does still apply). The other main point of interest is that a new subclass `TriangularMatrix` has been introduced. Since cholesky factors are triangular, `CholeskyFactor` has been subclassed to `TriangularMatrix` in Figure 10.

Figures 9 and 10 represent the same information but are structurally different. This section will show how to use transformations to show that Figure 10 is a refactoring of Figure 9, or equivalently to transform Figure 9 into Figure 10. Renaming has not taken place in this example, so the mapping ϕ is the identity. Difference matching proceeds as follows.

Step I: match class structures After applying all normal rules (*inheritAttr*, *inheritOp* and *inheritConstr*) to both class diagrams under consideration (which makes explicit all the attributes and constraints in the diagrams), according to the difference matching procedure, the first step should be to match the class structures of the two class diagrams. By applying *introduceSubclass* repeatedly, each of the matrix subclasses from Figure 10 that do not appear in Figure 9 can be introduced into Figure 9 along with their appropriate constraints. Normal rules can then be applied so that each of these new subclasses in Figure 9 inherit the attributes, operations and constraints from class `Matrix`.

Step II: match associations (1) The new subclasses, `SymmetricDefinite-NonNegMatrix` (SND) and `SymmetricDefinitePositiveMatrix` (SPD), introduced into Figure 9 in Step I do not have the same associations as the corresponding classes in Figure 10. Since the subclasses were introduced by the *introduceSubclass* rule, they currently have only a single association — a generalization from themselves to `Matrix`. Associations belonging to `Matrix` can be introduced, however, using *transitivityAssocs* to propagate the `choleskyFactors` and `rankOneFactors` associations resulting in the model fragment in Figure 13. Note that the multiplicities in Figures 10 and 13 do not quite match. However, since *inheritConstr* was applied exhaustively in Step I, SND and SPD both have the constraints Φ_1 and Φ_2. Using the rule *inv2assoc*, Φ_1 and Φ_2 can be replaced by updating the multiplicities on these associations. This results in the multiplicities at the `CholeskyFactor` end of SND becoming 2 rather than $0, 2$, and similarly at the `RankOneFactor` end of SPD. On the other hand, the closed world assumption (see section 2.2) allows for the removal of the `choleskyFactors` association from SPD and the `rankOneFactors` association from SND (since they are uni-directional associations with a zero multiplicity at their navigable end). In a similar way, the `choleskyFactors` and `rankOneFactors` associations and constraints Φ_1 and Φ_2 can be propagated to all other subclasses of `Matrix`. In each of these cases, both associations are eliminated.

Step III: match associations (2) The final step is to introduce the inheritance link from `CholeskyFactor` to `TriangularMatrix` in the transformed version of Figure 9. This can be done using the rule *inferGen*, since the matrices have the same operations and attributes, all constraints of `TriangularMatrix` hold in `CholeskyFactor`, and all associations of `TriangularMatrix` are also associations of `CholeskyFactor`.

The transformed version of 9 and 10 now look the same, and hence it has been shown that the two matrix models are refactorings of each other.

4 Conclusions and Further Work

This paper has presented a number of concrete examples of transformations over the Unified Modeling Language. This is the first paper (of which the author is aware) that has considered how to automate such transformations and has done so in the context of checking that two class diagrams (with constraints) are refactorings of each other. Although full automation is still the subject of research, a semi-automated procedure has been presented, and demonstrated on a real example. The notion of correctness of UML transformations used in this research is based on that given in [6] but it has not been shown here that each individual transformation is correct in this sense. Rather, the focus is on moving towards automation of the application of transformations for a suitable domain. Other authors are also considering the notion of transformation correctness (e.g., [7]) and it is likely that their techniques will be useful in this instance.

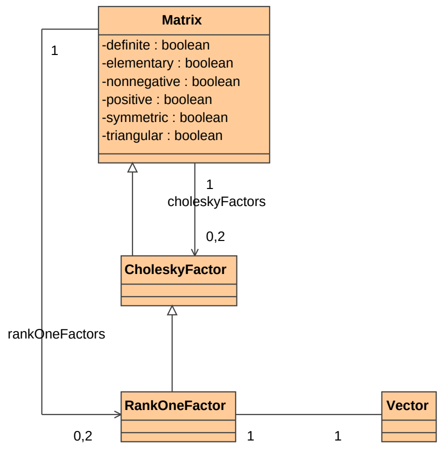


Fig. 9. Matrix Domain Model Before Refactoring.

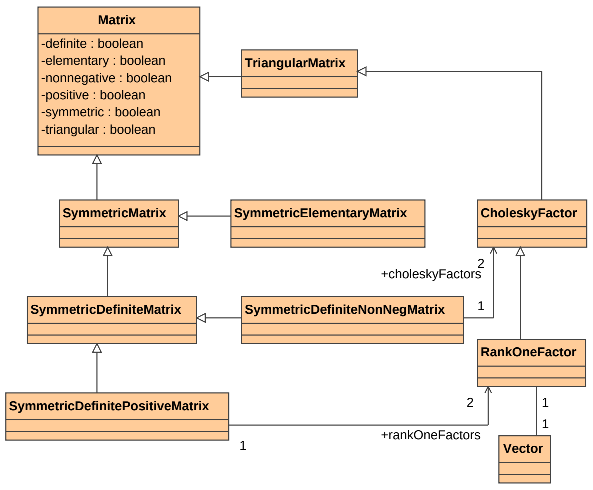


Fig. 10. Matrix Domain Model After Refactoring.

Φ_1 **context** `Matrix` **inv:**
```
 if symmetric=true  and nonnegative=true
                       and definite=true
  then choleskyFactors->size=2
  else choleskyFactors->size=0  endif
```

Φ_2 **context** `Matrix` **inv:**
```
 if symmetric=true  and positive=true
                       and definite=true
  then rankOneFactors->size=2
  else choleskyFactors->size=0  endif
```

Φ_3 **context** `CholeskyFactor` **inv:**
`triangular=true`

Fig. 11. Constraints for Figure 9.

Φ_3 **as before**

Φ_4 **context** `TriangularMatrix` **inv:**
`triangular=true`

Φ_5 **context** `SymmetricMatrix` **inv:**
`symmetric=true`

Φ_6 **context** `SymmetricDefiniteMatrix` **inv:**
`symmetric=true` **and** `definite=true`

Φ_7 **context** `SymmetricElementaryMatrix` **inv:**
`symmetric=true` **and** `elementary=true`

Φ_8 **context** `SymmetricDefiniteNonNegMatrix` **inv:**
```
symmetric=true  and definite=true
                and nonnegative=true
```

Φ_9 **context** `SymmetricDefinitePositiveMatrix` **inv:**
```
symmetric=true  and definite=true
                and positive=true
```

Fig. 12. Constraints for Figure 10.

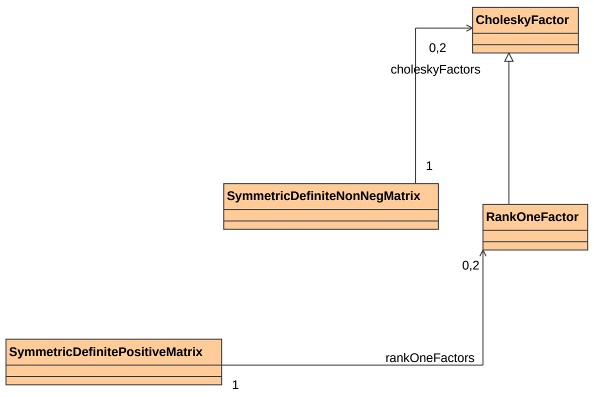


Fig. 13. Partially refactored matrix model.

References

1. J. Alemán, A. Toval, and J. Hoyos. Rigorously transforming UML class diagrams. In *Proceedings of the V Workshop MENHIR (Models, Environments and Tools for Requirements Engineering)*, Universidad de Granada, Spain, 2000.
2. Workshop on Automating Object Oriented Software Development Methods, June 2001. http://trese.cs.utwente.nl/ecoop01-aoom/.
3. J. Araújo, R. France, A. Toval, and J. Whittle. Workshop on Integration and Transformation of UML Models, June 2002. http://www-ctp.di.fct.unl.pt/ ja/wituml02.htm.
4. C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In *Proceedings of FMCAD'96*, November 1996.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001. To appear.
6. A. Evans. Reasoning with UML class diagrams. In *Workshop on Industrial Strength Formal Methods (WIFT98)*. IEEE Press, 1998.
7. R. France. A problem-oriented analysis of basic UML static requirements modeling concepts. *ACM SIGPLAN Notices*, 34(10):57–69, 1999.
8. M. Gogolla and M. Richters. Equivalence rules for UML class diagrams. In J. Bézivin and P.-A. Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998*, pages 87–96, 1998.
9. K. Lano and A. Evans. Rigorous development in UML. In *Fundamental Approaches to Software Engineering, FASE'99*. Springer-Verlag, 1999.
10. Unified Modeling Language specification version 1.4, September 2001. Available from The Object Management Group (http://www.omg.org).
11. A. Toval and J. Alemán. Formally modeling UML and its evolution: a holistic approach. In S. Smith and C. Talcott, editors, *Formal Methods for Open Object-based Distributed Systems IV*, pages 183–206, 2000.
12. J. Whittle and J. Schumann. Generating Statechart Designs From Scenarios. In *Proceedings of International Conference on Software Engineering (ICSE 2000)*, pages 314–323, Limerick, Ireland, June 2000.